# 3.4 Conditional Execution

So far, all of the function bodies we've written have consisted of a sequence of statements that always execute one after the other.[1] But sometimes we want to execute a statement or

[1] This kind of code block is sometimes called a "straight line program", since the statements form a linear path from one to the next.

block of statements only some of the time, based on some condition.

This is similar to the implication operator we saw when discussing propositional logic. The implication $p \Rightarrow q$ states that whenever $p$ is True, $q$ must also be True. In Python, what we would like to express is something of the form "Whenever $p$ is True, then the block of code `block1` must be executed". To do so, we'll introduce a new type of Python statement that play a role analogous to $\Rightarrow$ in propositional logic.

## *The if statement*

Python uses the **if statement** to express conditional execution of code. An if statement is a **compound statement**, meaning it contains other statements within it.[2] Here is our first

[2] Analogously, a expression like `3 + 4` is a *compound expression*, since it consists of smaller expressions (`3` and `4`).

syntax for an if statement:

```
if <condition>:
    <statement>
    ...
else:
    <statement>
    ...
```

The if statement uses two keywords, `if` and `else`.[3] The `<condition>` following `if` must be an

[3] Careful: we saw the `if` keyword used earlier to express conditions in comprehensions. The use of `if` here is logically similar, but quite different in how Python interprets it.

expression that evaluates to a boolean, called the **if condition**. This expression plays a role analogous to the hypothesis of an implication.

The statements on the lines after the `if` and `else` are indented to indicate that they are part of the `if` statement, similar to how a function docstring and body are indented relative to the function header. We call the statements under the `if` the **if branch** and the statements under the `else` the **else branch**.

When an `if` statement is executed, the following happens:

1. First, the if condition is evaluated, producing a boolean value.
2. If the condition evaluates to `True`, then the statements in the if branch are executed. If the condition evaluates to `False`, then the statements in the else branch are executed instead.

Let us consider an example. Suppose Toronto Pearson Airport (YYZ) has hired us to develop some software. The first feature they want is to show their clients if a flight is on time or delayed. The airport will provide us with both the time a flight is scheduled to depart and an estimated departure time based on the plane's current GPS location. Our task is to report a status (as a string) to display a string. Here is the function header and docstring:

```
def get_status(scheduled: int, estimated: int) -> str:
    """Return the flight status for the given scheduled and estimated
         departure times.

    The times are given as integers between 0 and 23 inclusive, representing
    the hour of the day.

    The status is either 'On time' or 'Delayed'.

    >>> get_status(10, 10)
    'On time'
    >>> get_status(10, 12)
    'Delayed'
    """
```

Now, if we only needed to calculate a `bool` for whether the flight is delayed, this function would be very straightforward: simply return `estimated <= scheduled`, i.e., whether the estimated departure time is before or at the scheduled departure time. Boolean expressions like this are often useful first steps in implementing functions to determine different "cases" of inputs, but they aren't the only step.

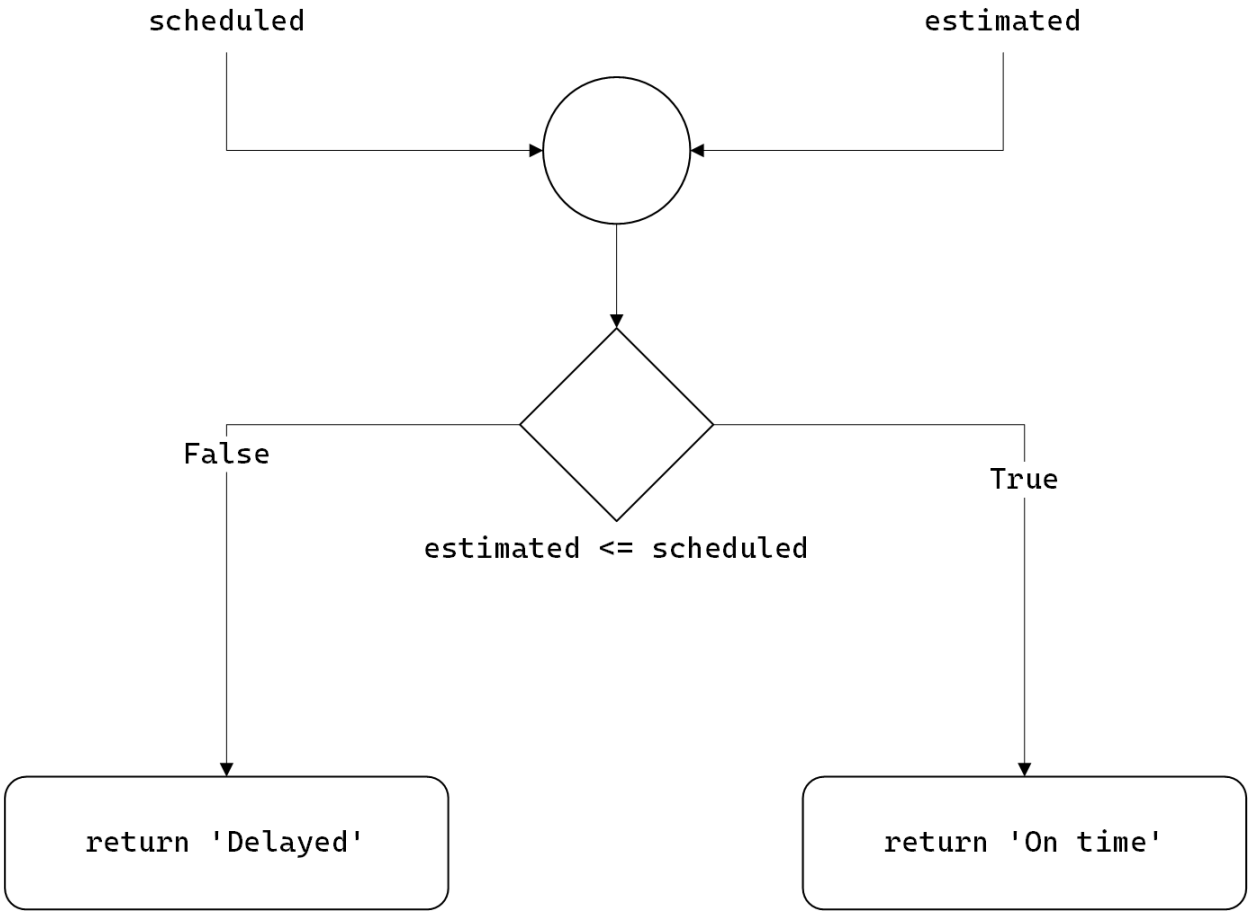
Instead, we use if statements to execute different code based on these cases. Here's our implementation of `get_status`:

```
def get_status(scheduled: int, estimated: int) -> str:
    """..."""
    if estimated <= scheduled:
        return 'On time'
    else:
        return 'Delayed'
```

Our if statement uses the boolean expression we identified earlier (`estimated <= scheduled`) to trigger different `return` statements to return the correct string.

### *A simple control flow diagram*

One useful tool for understanding if statements is drawing *control flow diagrams* to visualize the order in which statements execute. For example, here is a simple diagram for our `get_status` function above:




An `if` statement introduces a "fork in path" of a function's control flow, and this is why we use the term *branch* for each of the `if` and `else` blocks of code.

## *Code with more than two cases*

Now suppose Toronto Pearson Airport has changed the requirements of our feature. They've noticed that whenever a flight is delayed by more than four hours, the airline cancels the flight. They would like our `get_status` function to accommodate this change, so that the set of possible outputs is now `{'On time', 'Delayed', 'Cancelled'}`.

```
def get_status_v2(scheduled: int, estimated: int) -> str:
    """Return the flight status for the given scheduled and estimated
        departure times.

    The times are given as integers between 0 and 23 inclusive, representing
```

```
    the hour of the day.

    The status is 'On time', 'Delayed', or 'Cancelled'.

    >>> get_status_v2(10, 10)
    'On time'
    >>> get_status_v2(10, 12)
    'Delayed'
    >>> get_status_v2(10, 15)
    'Cancelled'
    """
```

Let's consider what's changed between this version and our previous one. If the estimated time is before the scheduled time, nothing's changed, and `'On time'` should still be returned. But when the estimated time is after the schedule time, we now need to distinguish between two separate subcases, based on the difference in time. We can express these subcases using nested if statements, i.e., one if statement contained in a branch of another:

```
def get_status_v2(scheduled: int, estimated: int) -> str:
    """..."""
    if estimated <= scheduled:
        return 'On time'
    else:
        if estimated - scheduled <= 4:
            return 'Delayed'
        else:
            return 'Cancelled'
```

This function body is correct, but just like with expressions, excessive nesting of statements can make code difficult to read and understand. So instead of using a nested if statement, we'll introduce a new form of if statement that makes use of the `elif` keyword, which is short for "else if".

```
if <condition1>:
    <statement>
    ...
elif <condition2>:
    <statement>
    ...
... # [any number of elif conditions and branches]
else:
    <statement>
    ...
```

When this form of if statement is executed, the following happens.

1. First, the if condition (`<condition1>`) is evaluated, producing a boolean value.
2. If the condition evaluates to `True`, then the statements in the if branch are executed. If the condition evaluates to `False`, then next elif condition is evaluated, producing a boolean.
3. If that condition evaluates to `True`, then the statements in that elif branch are executed. If that condition evaluates to `False`, then the next elif condition is evaluated. This step repeats until either one of the elif conditions evaluate to `True`, or all of the elif conditions have evaluated to `False`.
4. If neither the if condition nor any of the elif conditions evaluate to `True`, then the else branch executes.

Here is how we can use `elif` to rewrite `get_status` without nested if statements.

```
def get_status_v3(scheduled: int, estimated: int) -> str:
    """Return the flight status for the given scheduled and estimated
         departure times.

    The times are given as integers between 0 and 23 inclusive, representing
    the hour of the day.

    The status is 'On time', 'Delayed', or 'Cancelled'.

    >>> get_status_v3(10, 10)
    'On time'
    >>> get_status_v3(10, 12)
    'Delayed'
    >>> get_status_v3(10, 15)
    'Cancelled'
    """
    if estimated <= scheduled:
        return 'On time'
    elif estimated - scheduled <= 4:
        return 'Delayed'
    else:
        return 'Cancelled'
```

This code is logically equivalent to the previous version, but it's easier to read because there's no more nesting! Now, it is clear exactly what are the three possible branches of execution for this function.

## *Testing all the branches*

Adding branching to our control flow makes our functions more complex, and so we need to pay attention to how we test our code. With functions that contain if statements, any one particular input we give can only test one possible execution path, so we need to design our unit tests so that each possible execution path is used at least once. This form of test design is called **white box** testing, because we "see through the box" and therefore can

design tests based on the source code itself. In contrast, **black box** testing are tests created without any knowledge of the source code (so no knowledge of the different paths the code can take).

In our doctests for `get_status_v3`, we chose three different examples, each corresponding to a different possible case of the if statement. This was pretty straightforward because the code is relatively simple, but we'll study later example of more complex control flow where it won't be so simple to design test cases to cover each branch. In fact, the percentage of lines of program code that are executed when a set of tests for that program is called **code coverage**, and is a metric used to assess the quality of tests. While a set of tests may strive for 100% code coverage, this does not always occur as our programs grow in complexity. The concept of code coverage and other metrics used to evaluate tests is something we'll only touch on in this course, but in future courses you'll learn about this in more detail and even use some automated tools for calculating these metrics.[4]

[4] In particular, even though code coverage is a commonly used metric, it is also criticized for giving a false sense of quality of a test suite. Just because all lines of code are executed at least once does not actually mean that the tests chosen cover all possible cases to consider for a program. We'll see a simple example of this in the following section.

## *Building on our example*

Toronto Pearson Airport is beginning to trust us with more data, and are requesting more complex features as a result. They now want us to write a function that determines how many flights are cancelled in a day. The airport will provide us with the data as a dictionary (i.e., `dict`), where the keys are unique flight numbers and the values for each flight number is a two-element list. The first element is the scheduled time and the second element is the estimated time. More succinctly, the data is a mapping of the form: `{ flight_number: [scheduled, estimated] }`.

Unlike earlier, when our function input was only two integers, we are now working with a collection of data. Before we start trying to solve the problem, let's create some example data in the Python console. Specifically, we'll create a dictionary with values for three different Air Canada flight numbers.

```
>>> flights = {'AC110': [10, 12], 'AC321': [12, 19], 'AC999': [1, 1]}
>>> flights['AC110']
[10, 12]
```

We know that we can query the dictionary by providing an existing key. The value associated with a key is a list of integers, and we can index the list to retrieve those integers. Index 0 of the list refers to the flight number's scheduled time, while index 1 refers to the estimated time. Let us call our `get_status_v3` function for flight `'AC110'`:

```
>>> flight_ac110 = flights['AC110']
>>> get_status_v3(flight_ac110[0], flight_ac110[1])
```

```
    'Delayed'
```

We're making great progress! Instead of specifying the flight number ourselves (i.e., `'AC110'`), we would instead like to substitute in different flight numbers based on the data we receive from the airport. We can do that using comprehensions. Let's explore and see what we can get:

```
>>> {k for k in flights}
{'AC999', 'AC110', 'AC321'}
>>> {get_status_v3(flights[k][0], flights[k][1]) == 'Cancelled' for k in
        flights}
{False, True}
>>> [get_status_v3(flights[k][0], flights[k][1]) == 'Cancelled' for k in
        flights]
[False, True, False]
```

Our first set comprehension can get us the set of flight numbers, but that doesn't tell us if the flight was cancelled or not. When we created our second set comprehension we could see that there was at least one flight cancelled. Remember that sets only contain unique elements, and this set consists of all possible boolean values. When we create a list comprehension, we can see that exactly one out of three flights were cancelled (there is one `True` value). But remember that the airport only wants to know how many flights were cancelled; a single integer value. Currently, we have a list of boolean values.

Let us now try to combine the first set comprehension with the second, using the filtering we learned in the last section.

```
>>> {k for k in flights if get_status_v3(flights[k][0], flights[k][1]) ==
        'Cancelled'}
{'AC321'}
>>> [k for k in flights if get_status_v3(flights[k][0], flights[k][1]) ==
        'Cancelled']
['AC321']
```

Excellent! We now have a set of flight numbers that were cancelled. To convert this into an integer, we can use the built-in `len` function on the set.[5] Let's see what all this looks like in

[5] Something to think about: does it matter if we use the list or set comprehension here?

a function:

```
def count_cancelled(flights: dict) -> int:
    """Return the number of cancelled flights for the given flight data.

    flights is a dictionary where each key is a flight ID,
```

```
    and whose corresponding value is a list of two numbers, where the first
        is
    the scheduled departure time and the second is the estimated departure
        time.

    >>> count_cancelled({'AC110': [10, 12], 'AC321': [12, 19], 'AC999': [1,
        1]})
    1
    """
    cancelled_flights = {k for k in flights
                         if get_status_v3(flights[k][0], flights[k][1]) ==
        'Cancelled'}
    return len(cancelled_flights)
```

Let's review what we learned in this example:

- We can try to remember how we can use what we've learned by exploring in the Python console, well before starting to write the function. Here, we refreshed our memory on how we might look up values from dictionaries, index lists, call functions, create comprehensions, and filter collections.
- We can substitute in different values for a function's input using comprehensions.
- We can reuse functions we've already created and tested (like `get_status_v3`) to help implement other functions.